(12) UK Patent Application (19) GB (11) 2 201 015 (13) A

(43) Application published 17 Aug 1988

(21) Application No 8702919

(22) Date of filing 10 Feb 1987

(51) INT CL⁴
G06F 9/30 15/16

(52) Domestic classification (Edition J):
G4A MP PX

(56) Documents cited
GB A 2177526 GB A 1026889

(58) Field of search
G4A
Selected US specifications from IPC sub-class G06F

(71) Applicant
The University of Southampton

(Incorporated in United Kingdom)

Highfield, Southampton, SO9 5NH

(72) Inventors
Christopher Roger Jesshope
Jimmy Melvin Stewart
Russell John O'Gorman

(74) Agent and/or Address for Service
Reddie & Grose
16 Theobalds Road, London, WC1X 8PL

(54) Parallel processor array and array element

(57) A processor element (PE) for a parallel processor array comprises a memory 16, ALU 14, accumulator 12 and status register 20 for status or condition flags. This register is connected to an inactivity circuit 22 (Fig. 2) which includes an inactivity latch for each flag, forming a set (24 Fig. 2). The PE has a first port coupled to bus (64 Fig. 4) for instructions from the array controller which can render a PE inactive by setting a latch unconditionally or conditionally upon the status of the corresponding flag and which can reset a latch. When any latch is set, the PE does not execute normal instructions received from the controller on a second port. Because inactivity can be controlled conditionally on all status flags it is possible to implant various program constructs (IF THEN....ELSE; DO WHILE; etc.) as simply as in assembly language for a conventional serial processor with its own program counter, even although only the controller has such a counter, and such constructs are performed selectively by the PE's. The additional hardware complexity of each PE is small.

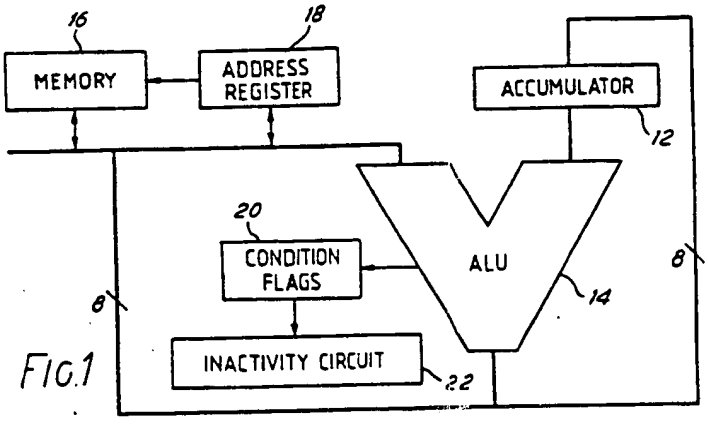

FIG.1

The drawing(s) originally filed was (were) informal and the print here reproduced is taken from a later filed formal copy.

GB 2 201 015 A

2201015

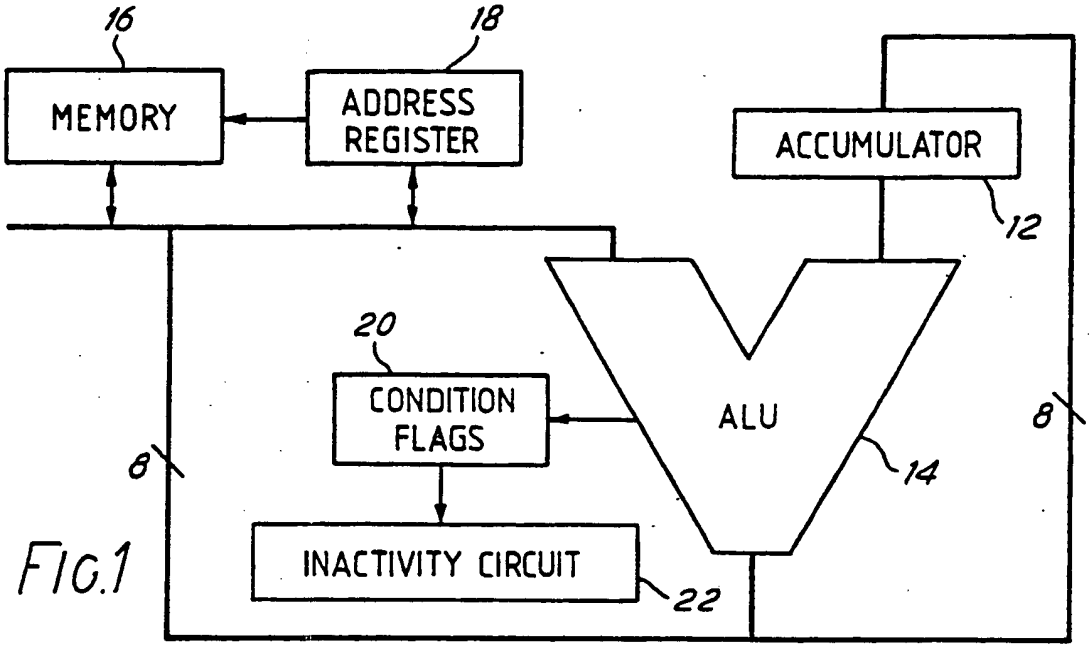

FIG.1

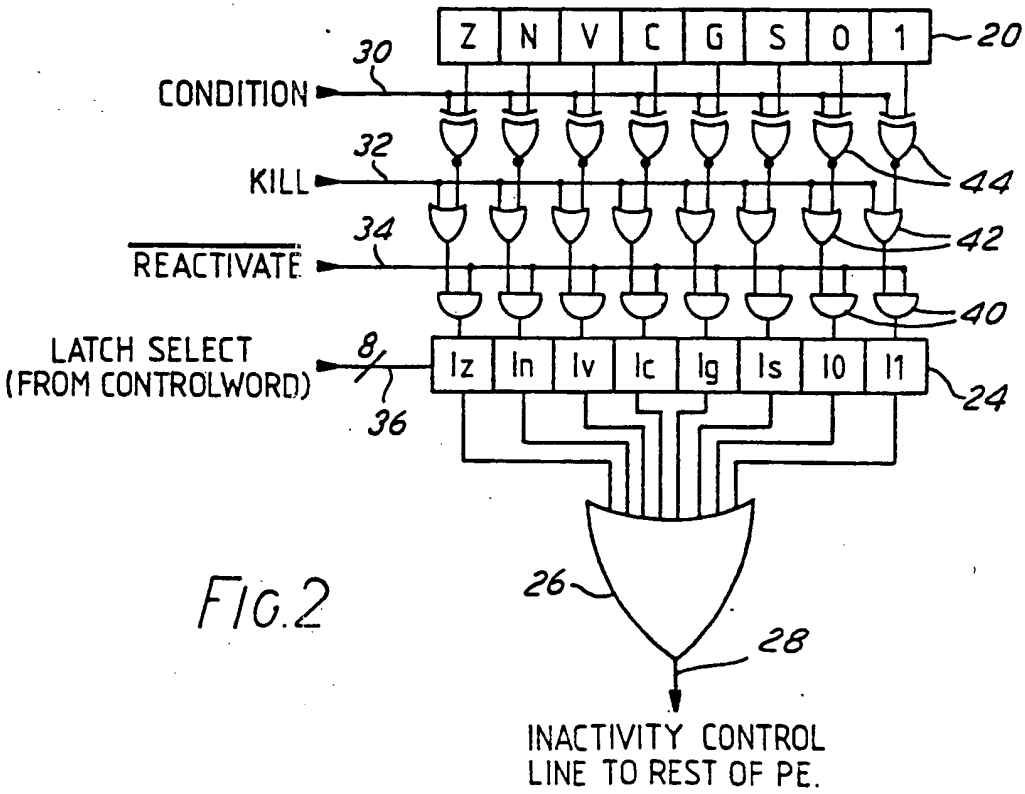

FIG.2

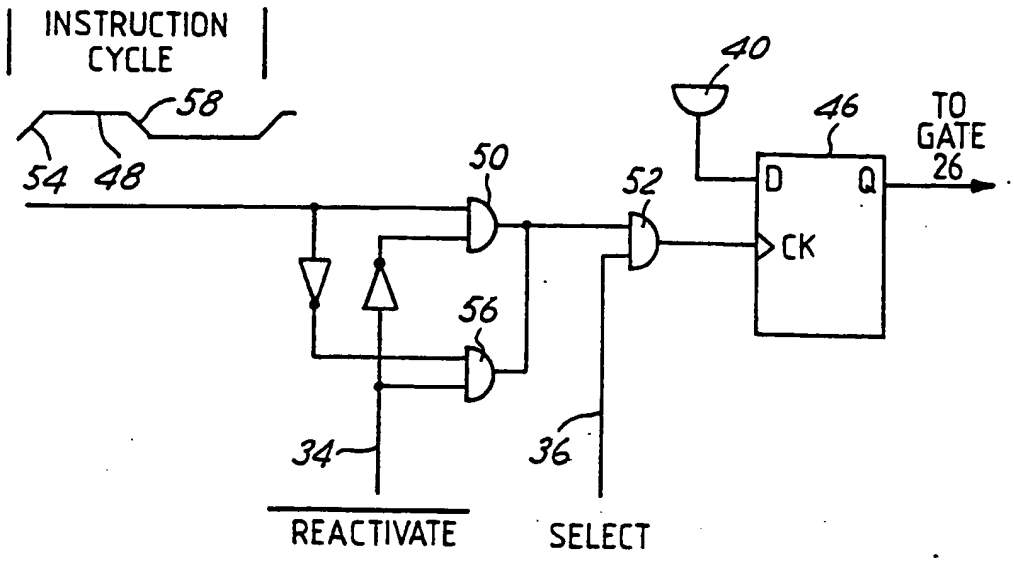
INSTRUCTION CYCLE
58
54
48
50
52
40
46
TO GATE 26
D
Q
CK
56
34
36
REACTIVATE
SELECT

FIG. 3

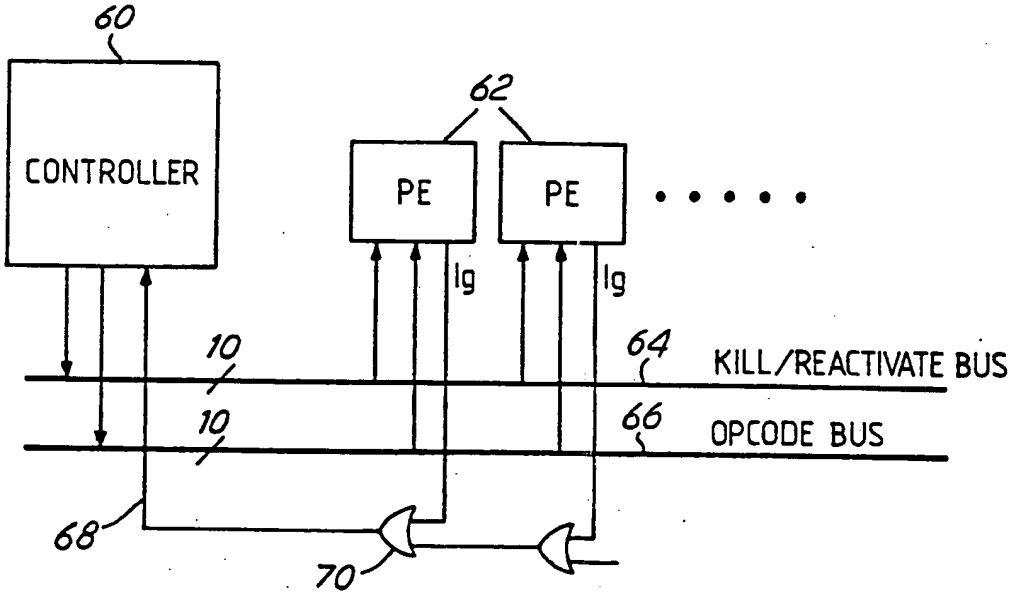
60
CONTROLLER
62
PE
PE
Ig
Ig
10
10
64
KILL/REACTIVATE BUS
66
OPCODE BUS
68
70

FIG. 4

2201015

PARALLEL PROCESSOR ARRAY AND ARRAY ELEMENT

The present invention relates to the control of parallel processor arrays consisting of a plurality of individual processor elements. In particular, the invention is concerned with achieving local data-dependent control within each processor element (PE).

In conventional single chip microprocessors, data-dependent control is achieved by means of BRANCH or JUMP instructions which are conditional on certain local conditions being met. An indication of the local condition is usually provided by a flag bit which is set or clear. The BRANCH or JUMP instructions are executed by the program sequencer part of the microprocessor which causes the program counter to be altered and, hence, the microprocessor to jump to a different part of the instruction set forming the program.

In single instruction, multiple data (SIMD) processor arrays, one instruction is sent by the controller to each of a number of processor elements forming the array. Each processor element executes the instruction on its own unique stream of data. Data-dependent conditions will inevitably differ in different processor elements and this fact makes control at a global level difficult, if not impossible, to achieve because a program branch address common to all processor elements does not exist.

It is known to provide a processor element of an SIMD array with an activity control register storing a flag which may be set and reset to disable or "kill" the processor element and to reactivate the element. The setting and resetting takes place conditionally, but the system is no-where near as flexible in controlling program flow as branch and jump instructions in a serial (SISD) processor.

The object of the present invention is to provide a processor element which can be used in an SIMD or other parallel processor array to render control as flexible and easy to use as the conditional statements in a conventional microprocessor. It is, however, essential that any control scheme provided for such an array should not rely on the program counter being altered as the processor elements in the array are not provided with individual program counters.

Processor arrays achieve high data processing rates by

replicating many (e.g. 1024) identical processing elements. It is, therefore, also highly desirable to keep the complexity of each processor element as low as possible in order to avoid additional costs involved in replicating complex circuitry in each processor element.

In accordance with the invention there is provided a processor element for a parallel processor array, comprising a plurality of inactivity latches corresponding to a plurality of condition flags and each of which is settable and resettable in dependence upon the status of the corresponding condition flag, and serving, when set to disable activities of the processor element, and a control circuit responsive to received instructions in conjunction with condition flags of the processor element to set and reset the latches.

Thus, in a processor array formed from such elements, the controller cycles through all possible instructions in strict sequence. The instructions must cover every possible condition which could arise in the processor elements of the array but each processor element discriminates, by reference to its inactivity latches, between those instructions which are to be executed on its own data and those intended for other processor elements of the array.

The control scheme described above gives the flexibility of a conventional microprocessor utilising relatively simple circuitry in the individual processor elements so that hardware costs are kept to a minimum and use of resources is maximised. It further allows each processor element in a SIMD processor array to make conditional forward 'branch' operations on the basis of conditions generated locally, because the instructions up until the relevant label are ignored. Such 'branch' operations appear to the user to be analogous to conditional instructions in a conventional single chip microprocessor.

The invention will now be described in more detail, by way of example, with reference to the drawings, in which:

Fig. 1 is a block diagram of a processor element embodying the invention,

Fig. 2 illustrates an inactivity circuit of the processor element in detail,

Fig. 3 shows the circuit of a single inactivity bit latch, and Fig. 4 is a block diagram of an SIMD processor array embodying the invention.

A conventional microprocessor incorporates an arithmetic and logic unit (ALU) and a program sequencer. The sequencer allows conditional branching and loops to be performed by altering the program counter, as mentioned above. In a SIMD array, the only program counter is that in the main controller and, consequently conditionsl JUMP and CALL instructions, which cause the program sequencer to skip to an entirely different program address, are not possible. In any event, conditions vary from one part of the array to another, so that evaluation of a given condition will vary from one processor element to another. Each processor element must, therefore, be provided with means by which local conditional statements can be evaluated without reference to a central program counter.

The processor element shown in Fig. 1 is based on a conventional 8 bit accumulator architecture and includes an accumulator 12, an ALU 14, a memory 16 and an address register 18. The ALU permits ADD, SUB, AND, OR, COMPARE and EOR operations between the memory 16 and the accumulator 12. Operations which can be performed on the address register 18 are INC and DEC. The address can be set from the accumulator 12 or from the array's central controller. The central controller broadcasts the same instruction to all the processor elements of the array and supplies a common 8-bit literal field. The processor element does not include a conventional set of registers but a status register 20 is provided for a set of eight states or condition flags.

As thus far described, there are no differences between the processor element shown and a conventional microprocessor. The processor element is, however, additionally provided with an inactivity control circuit 22, including a set of eight single bit local inactivity latches 24 (Fig.2). Fig. 2 also shows the status register 20 and circuitry, described below, enabling the latches 24 to be individually set and reset. The cutputs of the latches are combined in an OR gate 26 whose output 28 is an inactivity control line. When any one or more of the latches 24 is set and the line

28 is high, the processor element is disabled from implementing instructions, e.g. by disabling storage elements within the processor element. Disabling is also referred to herein as "killing".

The flags and latch bits listed in Table 1 below comprise both conventional and novel conditions.

Table 1

Condition	Flag	Latch Bit
Zero	Z	lz
Negative	N	ln
Carry	C	lc
Overflow	V	lv
Global	G	lg
Sticky	S	ls
User Zero	0	l0
User One	1	l1

The global condition is explained below. The sticky flag, once set to logical 1, will remain set until an explicit "clear flag" instruction is performed by the processor element. This operation is useful when performing normalizing functions on floating point mantissae. The two user flags are provided to allow the user to implement further logical functions without being tied to in-built conditions such as zero, negative, carry and overflow. These user flags are optional but may be provided up to the limit set by the number of control lines from the controller to the processor elements.

Reverting to Fig.2, the latch bits are set and reset under the control of a condition (positive or negative) line 30, an unconditional KILL line 32, an inverted REACTIVATE line 34 and an 8-bit latch select bus 36. The signals on these lines are determined by 10-bit kill/reactivate fields broadcast to the processor elements by the controller and comprising a two-bit operation code and an 8-bit latch select code. The operation codes are set out in Table 2, which also shows the decoding to the lines 30, 32 and 34.

Table 2

Operation	Code	Condition Line	Kill Line	Reactivate-
Kill unconditionally	00	X	1	0
Kill on positive condition	01	1	0	0
Kill on negative condition	10	0	0	0
Reactivate	11	X	X	1

X means "don't care".

If the code is 11 and reactivate is true, the inverted reactivate line 34 is false and the outputs of all of eight AND gates 40 are false. These false outputs reset those of the latches 24 whose latch select lines 36 are high, as will be explained with reference to Fig. 3.

If the code is 00 the kill line 32 is high and the outputs of eight OR gates 42 will be high. Since the inverted reactivate line 34 is also high (Table 2), the outputs of the AND gates 40 are true and set those of the latches 24 who latch select lines 36 are high.

If the code is 01 or 10, the outputs of some of a set of eight exclusive NOR gates 44 will be high, depending upon the states of the flags in the register 20. The outputs of only the corresponding AND gates 40 will be true and a true output will set the corresponding latch if its latch select line 36 is high.

The circuit of a single latch is shown in Fig.3 and comprises a D-type flip-flop 46 with its D input connected to the output of the corresponding AND gate 40. The flip-flop is clocked on a positive-going edge on its CK terminal. A clock signal 48 is passed by an AND gate 50 when reactivate is true and is passed to the CK terminal by an AND gate 52 only if the corresponding latch select line 36 is true. Accordingly, a processor element will be reactivated at the start of an instruction cycle, in response to the leading edge 54 of the clock signal 48. The inverted clock signal 48 is passed by an AND gate 56 when reactivate is false and is passed to the CK terminal by the AND gate 52 only if the corresponding latch select

line 36 is true. Accordingly, a processor element will be killed on the falling edge 58 at the end of the first phase of the clock cycle.

The effect of this is that a processor element (PE) will always execute an instruction which is broadcast in parallel with a kill or reactive instruction which takes effect on that element.

The latch select lines 36 directly copy the 8-bit latch select code. Fig.4 illustrates a fragment of an array processor comprising a controller 60 and a plurality of processor elements 62 to which the controller 60 broadcasts the above-described 10-bit kill/reactivate fields on a bus 64 and the 10-bit operation codes, on a bus 66, for the other processing operations. In a practical embodiment there may be 1024 processor elements.

The processor element thus has separate ports for the kill/reactivate fields and the normal operation codes, so that kill/reactivate instructions can be broadcast in parallel with normal operation codes. When a PE is rendered inactive, it is disabled as a processing entity, that is to say it is not influenced by instructions received on its normal opcode port, but the circuitry (Fig.2) on the bill/reactivate port remains functional.

The processor element described above can be controlled by 10-bit fields corresponding to the following assembly language kill instructions in Tables 4 to 6.

Table 4

"Branch on Positive Conditions

KZS	LABEL	Deactivate if zero flag set
KCS	LABEL	Deactivate if carry flag set
KMI	LABEL	Deactivate if negative flag set
KVS	LABEL	Deactivate if overflow flag set
KGS	LABEL	Deactivate if global flag set
KSS	LABEL	Deactivate if sticky flag set
KOS	LABEL	Deactivate if first user flag set
K1S	LABEL	Deactivate if second user flag set

Table 5

"Branch on Negative Conditions

KNZ	LABEL	Deactivate if zero flag not set
KCC	LABEL	Deactivate if carry flag not set
KPL	LABEL	Deactivate if negative flag not set
KVC	LABEL	Deactivate if overflow flag not set
KGC	LABEL	Deactivate if global flag not set
KSC	LABEL	Deactivate if sticky flag not set
KDC	LABEL	Deactivate if first user flag not set
K1C	LABEL	Deactivate if second user flag not set

Table 6

Unconditional "Branches"

KKZ	LABEL	Set zero latch bit lz
KKC	LABEL	Set carry latch bit lc
KKN	LABEL	Set negative latch bit ln
KKV	LABEL	Set overflow latch bit lv
KKG	LABEL	Set global latch bit lg
KKS	LABEL	Set sticky latch bit ls
KK0	LABEL	Set first user latch bit l0
KK1	LABEL	Set second user latch bit l1

and Fig. 2.

From Tables 2 and 4 and Fig.2 it can be seen, that the kill/reactivate field for KVS will be

0 1 0 0 1 0 0 0 0 0

since the most significant bits 01 identify the operation "kill on positive condition" and the sixth least significant bit identifies the overflow latch (Fig.2).

All the assembler instructions in Tables 4, 5 and 6 include LABEL. This label is not passed to the processor elements. The controller 60 automatically generates the appropriate reactivate command when it reaches the label. Those commands are given in Table 7, although they are never coded by the programmer, and they complement the unconditional kill commands of Table 6.

Table 7

Reactivate Commands

RRZ	Reset zero latch bit lz
RRC	Reset carry latch bit lc
RAN	Reset negative latch bit ln
RRV	Reset global latch bit lg
RRS	Reset sticky latch bit ls
RR0	Reset first user latch bit l0
RR1	Reset second user latch bit l1

The use of these instructions will now be clarified by looking at some examples:

Example 1

```
1)       LDA      Y
2)       ADD      £5        ; do something
3)       KZS      LABEL     ; Kill operation if zero flag is set
                            ; until LABEL reached
4)       ...               ; Section of code
5)       ...
6)       ...
7)LABEL  ...               ; PEs are reactivated by RRZ and this
                            ; instruction is executed
8)       ...
```

In example 1, all processing elements with their zero flags set at line 3 are disabled until line 7. At line 7 the PEs are reactivated in parallel with the instruction on that line and the instruction on line 7 is executed. The label specified on line 3 may be used by more than one Kill instruction. When the label is reached, all the PEs deactivated with reference to this label will be simultaneously reactivated. Note that the instruction RRZ to do this is inserted in the object code automatically by the assembler.

Processing elements may be disabled at different places by the SAME condition, but will always be reactivated at the same point, as in the next example.

Example 2

```
1)      ...
2)      KZS       LABEL
3)      ...
4)      ...
5)      KZS       LABEL
6)      ...
7)LABEL ...
8)      ...
```

Although Example 2 is legal, some combinations of deactivation and reactivation are not allowed. Here are two examples of illegal constructs:
The assembler will not allow a program to 'leap frog' on the SAME condition as shown in the next example.

Example 3

```
1)        ...
2)        KZS    LABEL 1
3)        ...
4)        ...
5)        KZS    LABEL 2
6)        ...
7)LABEL 1 ...              ;Reactivated here but this code
                           ;section is still dependent on
                           ;condition in line 5.
8)        ...
9)LABEL 2 ...
```

Nested deactivation on the SAME condition is also not allowed.

Example 4

```
1)          ...
2)          KZS     LABEL 1
3)          ...
4)          ...
5)          KZS     LABEL 2
6)          ...
7)LABEL 2   ...             ;Reactivated here but this code
                            ;section is still dependent on
                            ;condition in line 2.
8)          ...
9)LABEL 1   ...
```

'Leap frogging' and nested deactivation is possible if DIFFERENT conditions are used or if the condition is transferred to a different register in order to do the deactivation. Example 5 shows legal 'leap frogging' using zero and carry flags while Example 6 shows legal 'leap frogging' on a single condition (zero flag).

Example 5

```
1)          ...
2)          KZS     LABEL 1
3)          ...
4)          ...
5)          KCS     LABEL 2
6)          ...
7)LABEL 1
8)          ...
9)LABEL 2   ...
```

Example 6

```
1)          ...
2)          KZS     LABEL 1
3)          ...
4)          move Z to C
5)          KCS     LABEL 2
6)          ...
7)LABEL 1   ...
8)          ...
9)LABEL 2   ...
```

Although legal, examples 5 and 6 are not recommended as good programming practice.

Example 7 shows how a locally controlled IF THEN ELSE can be implemented. The construct required is:

```
IF accumulator  =0 THEN
        begin
                list of instructions
        end
ELSE
        begin
                list of instructions
        end
```

In assembler this becomes:

Example 7

```
                ;Zero flag set on the accumulator being loaded.
KNZ     ELSE    ;If zero flag not set, then turn off PE whilst
                ;the THEN statements are being executed
...             ;Begin THEN instructions
...
...                 ;Last 'THEN' statement
KKO     END         ;Jump to end of else statements.
                ;Unconditional deactivation using the 1st userflag.
                ;This deactivation is however activity controlled.
                ;If the zero flag was not set previously then this
                ;instruction is not performed.
```

```
ELSE ...          ;Start of ELSE statements
     ...
     ...
END  ...          ;Last 'ELSE' statement
                  ;All reactivation is done in parallel with this
                  ;instruction.
```

A WHILE construct is slightly more complex than the IF THEN ELSE. This is because the number of loops performed in WHILE construct is locally contrtolled but the controller must be aware when all the loops being executed locally are complete.

There are two ways of doing this. The first is to specify in code the maximum number of times the controller loops through the section of code where the WHILE construct is defined. After each iteration the processors that have satisfied the WHILE condition are disabled. The number of loops specified to the controller must be enough to allow all local conditions to be satisfied.

The second method does a global 'IF ANY' operation after each iteration. The controller checks a single bit that comes from the array. This bit is called the 'GLOBAL BIT' and is hardwired OR of all the global latch bits output by the PEs. The controller can then act conditionally on this bit. Examples of each method are given in Examples 8 and 9 respectively. The indented instructions in these examples refer to the array of PEs and the outer instructions are controller instructions. No instruction which relies on the program counter being altered, such as a FOR or BRANCH instruction, can be executed by the array, as previously explained.

The generation of GLOBAL BIT 68 is illustrated in Fig.4. OR gates 70 are used to represent the wired OR function which is performed on all global latch bits lg as output from the individual PE's.

We want to execute the following construct locally:

```
WHILE accumulator = 0
   DO
     Begin
          List of Instructions
     End
```

Example 8

```
      FOR MAX          ;start of controller loop
        KNZ  END       ;If the accumulator is not zero then kill
                       ;any activity until the end
        ...            ;Instructions that are to be performed within
        ...            ;the WHILE loop
        ...
        ...
      NEXT             ;end of controller loop
END   ...
      ...
```

Example 9

```
      SET GLOBAL;  Controller explicitly sets all global flags.
      WHILE global bit TRUE;            Controller construct.
        begin
         KNZ       END  ;condition tested at start of loop
         ...            ;do the instructions with the white loop
         ...            ;end of loop condition evaluated
         ...            ;e.g. Z flag false
         ...            ;move Z flag to global flag
                        ;this may need more than one instruction
        end             ;end of the loop (may need more than one
                         instruction)
                        ;to allow for propagation
END     ...             ;Reactivation here
        ...
```

Both the above implementation of the local condition WHILE loop have their advantages and disadvantages. The first method is better if MAX is not very great or if MAX normally occurs anyway. The disadvantage is that MAX must be finite and must be known at assembly time.

In the second method MAX does not have to be known and the loop will terminate as soon as all the PEs are finished. The looping overhead in the second method is, however, greater because the condition bit must first be moved to the global latch and then a cycle must be allowed for the contents of the global latch bits to propagate out so as to form the new global bit for the controller to

test at the start of the loop.

The global flag can be thought of as a special example of a user flag, with the difference that the global latch bit are OK'd to form the global bit back to the controller.

To perform a DO WHILE construct the flag test is simply made the last instruction in the loop, hence ensuring the instructions in the loop are executed at least once.

With the IF THEN ELSE and WHILE loop illustrated above, any local construct can be implemented.

eg.

```
FOR:        for i = 1 to 100 do
              begin
                 ...
                 ...
              end

   becomes
            i = 1
            while i   = 100 do
                 begin
                   ...
                   ...
                   i = i + 1
                 end

REPEAT UNTIL:
                 repeat
                   ...
                   ...
                   ...
                 until x true
   becomes
                 do
                  ...
                  ...
                  ...
                 while x true
```

CASE

```
case x of
  1 :    ......
  2 :    ......
  3 :    ......
  4 :    ......
  otherwise :   ......
end
```

becomes

```
if x = 1 then   statements, set flag
if x = 2 then   statements, set flag
if x = 3 then   statements, set flag
if x = 4 then   statements, set flag
if flag not set then   ........
```

Note, however, in a sequential machine the machine can ignore the rest of the case statement as soon as a true condition is found. For this reason it is not efficient to implement the CASE construct as a series of IF statements in a sequential machine. In a SIMD machine, however, all the conditions MUST be evaluated because different cases will be found across the array. Premature termination of the case statement in a SIMD array is only possible if the flag which is set in each IF statement in the example above, is inverted and placed in the global bit register. The global bit will then inform the controller if there are 'ANY' cases still to be resolved. In all cases involving global control, it is only if the amount of conditional processing is large that a check on the global bit should be made.

Conditional CALL instructions may be done by disabling all the processing elements not wishing to execute the subroutine, before the microcontroller vectors to the new location in program memory. The programmer does, however, HAVE to be aware of which inactivity registers are used within the subroutine so that the same inactivity registers are not used to disable PE's prior to subroutine entry. This problem can be avoided by observing a simple rule on inactivity register usage. This might be that

subroutines are only permitted to use ALU FLAG inactivity registers; while the program uses only USER FLAG inactivity registers to control conditional execution of subroutines.

The status or condition flags of the register 20 may be manipulated as well as set or reset in dependence upon the outcome of operations. The instruction to manipulate the flags can be INVERT, SET, RESET and normal shift operations. Individual bits in the accumulator 12 can be transferred to specified flags.

As well as the simple operations set out in Tables 4 to 7 it is possible to code multiple operations whereby a plurality of latch bits are set or reset simultaneously, i.e. by including more than one set bit in the 8-bit latch selection field. If all eight bits are set, for example, it will be possible to effect global reactivation, i.e. clear all latch bits by broadcasting the kill/reactivate field

1 1 1 1 1 1 1 1 1 .

The embodiment of the processor element described above is based upon a conventional 8-bit, memory reference architecture. The invention is clearly not restricted to such an implementation. Different word sizes and/or architectures may be employed.

CLAIMS:

1. A processor element for a parallel processor array, comprising a plurality of inactivity latches corresponding to a plurality of condition flags and each of which is settable and resettable in dependence upon the status of the corresponding condition flag, and serving, when set to disable activities of the processor element, and a control circuit responsive to received instructions in conjunction with condition flags of the processor element to set and reset the latches.

2. A processor element according to claim 1, comprising means forming a disabling signal for the preocessor element as an OR function of the states of the inactivity latches.

3. A processor element according to claim 1 or 2, comprising at least one inactivity latch corresponding to a user flag whose status is settable by the user under program control.

4. A processor element according to claim 1, 2 or 3, comprising a first port for the aforesaid instructions and a second port for other instructions, the said control circuit being connected to the first port and remaining active when activities of the processor element are disabled, whereas the processor element is not influenced by instructions received on the second port when activities of the processor element are disabled.

5. A processor element according to claim 4, wherein each inactivity latch is reset at an early time in an instruction cycle, in response to one of the first said instructions, and is set at a later time in an instruction cycle, in response to one of the first said instructions, whereby the processor element will execute one of the said other instructions received on its second port simultaneously with any first said instruction on the first port whose effect is either to set any one or more of the inactivity latches or to reset every one or more of the inactivity latches which were in their set state.

6. A processor element according to claim 1 and substantially as hereinbefore described and as illustrated in Figs. 1 to 3 of the accompanying drawings.

7. A parallel processor array comprising a plurality of processor elements according to any of claims 1 to 6 and a controller adapted to broadcast to the elements both first instructions for setting and resetting the inactivity latches of the processor elements and other instructions.

8. A parallel processor array according to claim 7, wherein the controller is connected to the processor elements for simultaneously broadcasting one of the first instructions and one of the other instructions.

9. A parallel processor array according to claim 7 or 8, wherein one inactivity latch of each processor element is connected to an OR circuit returning a global signal to the controller.

Published 1988 at The Patent Office, State House, 66/71 High Holborn, London WC1R 4TP. Further copies may be obtained from The Patent Office, Sales Branch, St Mary Cr[illegible], Mary Cray, Kent, Con. 1/87.